Universidad del Valle de Guatemala

Algoritmos y Estructura de Datos

Jorge Eduardo Santizo Hernández (231926)

**Hoja de trabajo 6**

**Selección de Colección JCF para los Pokémon del Usuario**

Para almacenar la colección personalizada de Pokémon del usuario, he seleccionado un HashSet por las siguientes razones:

* Unicidad garantizada: El enunciado especifica que no se deben permitir Pokémon duplicados en la colección del usuario. HashSet garantiza la unicidad de elementos de forma intrínseca.
* Operaciones de alta eficiencia: Las operaciones críticas para esta aplicación tienen complejidad O(1) en promedio:
  + Verificar si un Pokémon existe en la colección: O(1)
  + Agregar un nuevo Pokémon: O(1)
  + Eliminar un Pokémon: O(1)
* Almacenamiento en disco: Para persistir la colección, simplemente se necesita serializar el HashSet, lo cual es directo y eficiente.

Comparación con otras estructuras

* ArrayList: Aunque permitiría mantener un orden de inserción, la verificación de existencia sería O(n), lo que reduce la eficiencia al agregar nuevos Pokémon.
* TreeSet: Mantendría los elementos ordenados automáticamente, pero con un costo de O(log n) para inserción y búsqueda, siendo innecesario si el ordenamiento no es una prioridad para la colección personal.
* LinkedHashSet: Sería apropiado si quisiéramos mantener el orden de inserción, pero con una pequeña sobrecarga de memoria respecto a HashSet.

Operaciones soportadas útiles para la hoja de trabajo

* add(Pokemon): Agregar un Pokémon si no existe
* contains(Pokemon): Verificar si un Pokémon ya existe en la colección
* remove(Pokemon): Eliminar un Pokémon de la colección
* stream(): Permitir operaciones como filtrado y ordenamiento cuando sea necesario mostrar la colección ordenada por tipo1

**Análisis de Complejidad para la Operación #4**

La operación #4 requiere mostrar el nombre y tipo1 de todos los pokémon existentes ordenados por tipo1.

Proceso de la operación

1. Recuperar todos los Pokémon del mapa
2. Ordenarlos por el campo tipo1
3. Mostrar el nombre y tipo1 ordenados

Análisis de complejidad por implementación:

HashMap

* Acceder a todos los elementos: O(n)
* Ordenar por tipo1: O(n log n)
* Mostrar datos ordenados: O(n)
* Complejidad total: O(n log n)

TreeMap

* Acceder a todos los elementos: O(n)
* Ordenar por tipo1: O(n log n) - requiere reordenamiento ya que están ordenados por nombre
* Mostrar datos ordenados: O(n)
* Complejidad total: O(n log n)

LinkedHashMap

* Acceder a todos los elementos: O(n)
* Ordenar por tipo1: O(n log n)
* Mostrar datos ordenados: O(n)
* Complejidad total: O(n log n)

Análisis de resultados

Gráfico, Gráfico de líneas

El contenido generado por IA puede ser incorrecto.

Tiempo promedio por implementación

* HashMap: 11,269,640 nanosegundos
* TreeMap: 10,174,360 nanosegundos
* LinkedHashMap: 9,272,090 nanosegundos

Teóricamente, todas las implementaciones tienen la misma complejidad O(n log n) para esta operación específica debido al paso de ordenamiento por tipo1, que determina la complejidad total.

Sin embargo, empíricamente LinkedHashMap demuestra ser la estructura más eficiente para esta operación particular con un tiempo promedio de ejecución de 9,272,090 nanosegundos.

Esto nos demuestra que, aunque la complejidad teórica puede ser la misma, los factores de implementación y características específicas de cada estructura pueden afectar significativamente el rendimiento práctico.

Link del repositorio: <https://github.com/San231926/ht6.git>